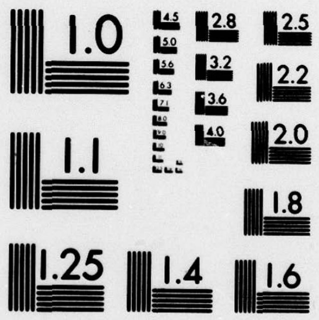
AD-A073 639 POLYTECHNIC INST OF NEW YORK BROOKLYN DEPT OF ELECTR--ETC F/G 9/2
SUMMARY OF TECHNICAL PROGRESS, INVESTIGATION OF SOFTWARE MODELS--ETC(U)
JUL 79 M L SHOOMAN, H RUSTON F30602-78-C-0057
UNCLASSIFIED POLY-EE-78-052 RADC-TR-79-188 NL

| OF |

AD
A073639

END
DATE
FILMED
10-79
DDC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(12) LEVEL II

RADC-TR-79-188
Interim Report
July 1979

SUMMARY OF TECHNICAL PROGRESS, INVESTIGATION OF SOFTWARE MODELS

Polytechnic Institute of New York

M. L. Shooman
H. Ruston

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 09 4 099

9 Interim technical rept.
1 Jan 78 - 31 Jan 79,

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-79-188	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 SUMMARY OF TECHNICAL PROGRESS, INVESTIGATION OF SOFTWARE MODELS.		5. TYPE OF REPORT & PERIOD COVERED Interim Technical Report Jan 78 - Jan 79
		6. PERFORMING ORG. REPORT NUMBER 14 POLY-EE-78-052
7. AUTHOR(s) 10 M. L. Shooman H. Ruston — Martin L./Shooman Henry /Ruston		8. CONTRACT OR GRANT NUMBER(s) 15 F30602-78-C-0057
9. PERFORMING ORGANIZATION NAME AND ADDRESS Polytechnic Institute of New York 333 Jay Street Brooklyn NY 11201		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 16 2304J401 17 J4
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE 11 July 1979 12 43p.
		13. NUMBER OF PAGES 44
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Alan N. Sukert (ISIS)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Software Testing; Software Test Drivers; Software Complexity; Program Complexity; Software Management Models		

DDC
RECEIVED
SEP 10 1979
B

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This interim report summarizes the research performed by Polytechnic Institute of New York for Rome Air Development Center under contract F30602-78-C-0057 for the period 1 Jan 78 to 31 Jan 79. The principal topics covered are (1) software test models and implementation of Automated test drivers to force-execute every program path, (2) development of new measures of program complexity based upon information theory, (3) models of software management and organizational structure, and (4) statistical measures relating the probability of finding a program error to the testing of that program.

DD FORM 1 JAN 73 1473 UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408 717 JOB

ABSTRACT

This interim report summarizes the research performed under Contract No. F30602-78-C-0057 by the Polytechnic Institute of New York for Rome Air Development Center from January 1, 1978 to January 31, 1979.

Recursive function theory was applied to the problem of program complexity. This study was completed and a technical report was issued. The present report contains the abstract of the technical reports.

The inquiry into the number of tests necessary to verify a computer program was undertaken. One phase of this study was completed, and a technical report was issued. The present report contains the abstract of the technical report.

A study was undertaken of software test models and of the implementation of associated test drivers. The present report describes this work as well as the test drivers obtained so far. A new approach to statistical aspects of program testing has been taken. The assumptions and the resulting model are described in the present report.

A new measure of complexity based upon information theory is introduced. This measure assumes that a language feature used infrequently is more likely to be used incorrectly than a language feature used frequently. The measure has the advantage of being sensitive to the different levels of nestings in either IF'S, DO'S, or procedures.

A number of different schemes are suggested for the calculation of the measure. A method for automatic calculation of the measure at an installation is also discussed.

The relation between program complexity and the program's information content was also investigated. The results obtained so far are described in this report.

Two models for the management of software were investigated. The first one models the productivity (measured in instructions per months), as well as the man-months required. The second model investigates different communication schemes that can be evolved when a problem is partitioned into several subproblems.

The concluding section of the report describes the planned work in the next period and lists professional activities of the personnel during the present reporting period.

SUMMARY OF TECHNICAL PROGRESS
SOFTWARE MODELING STUDIES
Table of Contents

List of Figures

List of Tables

1.0 INTRODUCTION

This interim report summarizes the research performed under Contract No. F30602-78-C-0057 by the Polytechnic Institute of New York for Rome Air Development Center from January 1, 1978 to January 31, 1979. The major topics investigated and summaries of the progress to date are described in Chapter 2.0; the references are listed in Chapter 3. In Chapter 4 the participating personnel are listed, as well as their activities during the reporting period. In Chapter 5 an outline is given of the planned direction of the research in the immediate future. Chapter 6 reports on the activities of the staff during 1978.

2.0 SUMMARY OF PROGRESS

This chapter summarizes the work performed. Upon the completion of each task a report on the task's results is issued. Several technical reports which document either a completed task or a phase in a continuing effort are in various stages of preparation.

2.1 STUDY OF RECURSIVE FUNCTION THEORY AND ITS APPLICATION TO PROGRAM COMPLEXITY -- by A.E. Laemmel

This study was completed and a technical report submitted in May of 1978 (revised version in October, 1978). The Abstract is presented below.

Abstract

It is desirable to have a measure of complexity for a computer program because more complex programs can be expected to cost more to produce, to require more time to test and debug, and to have more residual bugs even after testing and debugging. Presumably a program need be complex only to the extent that the problem it is solving is complex so that a way of calculating problem complexity might be expected to lead to a method of predicting programming and debugging costs. The purpose of this report is to examine the extent to which certain well-defined measures of complexity in recursive function theory might fulfill these expectation. The conclusion reached is that the complexity defined by recursive function theory (or an equivalent computability theory) is not too useful as a measure of programming difficulty.

Theories of computability. In the 1930's and 1940's mathematicians and logicians began to realize that certain problems which were very precisely stated were not just difficult but impossible to solve. In order to make such a strong assertion, the exact computational procedures which were to be allowed had to be explicitly described. Of course, if someone than discovered a more powerful procedure an "unsolvable" problem might well become solvable. However, although about half a dozen different models of computation were developed, they all lead to exactly the same class of solvable problems. This class is the general recursive functions in the terminology of one of the computation models due to Godel, Kleene, and others.

Degrees of computability Given that certain problems can be solved and others cannot, or that certain functions can be computed effectively while others cannot, can one then refine this go/no-go dichotomy to a scale of difficulties of computation for solvable problems? The answer turns out to be yes, especially if recursive function theory is taken as the basic model. In particular, the primitive recursive functions represent a more easily computed subset of the general recursive functions, and the Grzegorzyk classes partition the primitive recursive functions according to a certain definition of "difficulty of computation".

Relation to programming difficulties The following question now arises: Granted that a connection has been made between problem complexity and difficulty of computation, can this connection be extended to the difficulty of programming the computation which is to solve the problem? A case is made in this report for a "no" answer to this question.

The report reviews some theories of computational complexity, together with some estimates of their importance to software reliability. An attempt has been made to emphasize the practical rather than the theoretical. Examples are shown of very complex programs (in the sense of looping, time required, etc.) which should lead to few programming errors because the programming language allows a direct copying of the algorithm from a standard text book.

2.2 ON THE NUMBER OF TESTS NECESSARY TO VERIFY A COMPUTER PROGRAM -- by G.S. Popkin and M.L. Shooman

This study was completed and a report issued in June of 1978. The abstract is presented below.

Abstract

This report discusses various aspects of verifying that a computer program correctly carries out some specified functions. If the program was designed with the aid of a flowchart, the flowchart can be used to determine the number of tests necessary to verify the program. If the program was prepared without a flowchart, then either a flowchart or a directed graph must be prepared from the program to determine the number of tests.

One way to verify a computer program is to run it with sample input data and examine the results of correctness, i.e., test for agreement between the program output and the results that would be produced by correct execution of the specification on the same data. This requires prior determination via hand computation or other independent means of the correct outputs for the sample inputs. It is useful for us to first classify the different types of tests which we will be discussing. A level zero test is defined as a test consisting of one or more test cases which together cause every program statement to be executed at least once.

Before defining a level one test, it is necessary to define a program segment, which we shall do in terms of flowchart terminology. In general our flowcharts will consist of one start and one stop terminal (ovals), processing elements (rectangles or parallelograms), and decision elements also called deciders (diamond shape). A segment is any flow sequence:

a) from the START terminal to the first decider,

b) from the exit of one decider to the entry of another,

c) from the last decider to the STOP terminal,

whether or not any of these flows contain processing steps.

A subroutine is considered part of a segment if the segment contains a call to the subroutine. Segments may overlap, i.e., a processing step may be contained in more than one segment. If a program contains more than one TOP terminal, then a group of segments is formed by the flow to each terminal from the respective last decider.

In a level one test, all flowchart paths are force-traversed at least once. In a level two test every program path is naturally executed at least once. Level two and/or level three tests have formerly been called "exhaustive tests," but new definitions (given in the following section) are more precise and should replace the earlier usage. There are some particular types of problems, especially those dealing only with integer variables, that have a sufficiently limited set of input possibilities to make a level two or level three test practical. However, in general level two and three tests are intractable.

In this report, attention will be directed mainly to level one and related types of tests. That is, we will discuss tests which are oriented to the testing of program segments. Drawing on the work of M. Lipow (1), it will be shown how graphs, matrices, and zero-one integer linear programming (2) can be used to determine the number of test cases needed to perform a level one test, and the data needed to comprise those test cases.

The approach consists of finding the maximum incomparable set, i.e., the largest set of program segments through which one and only one test case should pass. The size of this set gives the minimum number of tests necessary to execute each segment at least once, while the elements of this set give the paths of each test.

In addition, a discussion is given of methods for estimating the number of paths in a loopless flowchart.

2.3 SOFTWARE TEST MODELS AND THE IMPLEMENTATION OF ASSOCIATED TEST DRIVERS -- by D.L. Baggi and M.L. Shooman

This study is nearing completion and a technical report is being prepared. Parts of the work in progress will be presented below.

2.3.1 Abstract

In the past most software tests were constructed by heuristics and drawing upon experience with similar software. Recently, enough preliminary work has been done to propose the analytical construction of test cases.

This report begins by defining five broad classes of software tests: Type 0, Type 1, Type 2, Type 3 and Type 4. Types 0,1, and 2 (i.e., level 0,1, and 2) were defined in the preceeding section. A Type 3 test is defined as an exhaustive interaction of all values of input and stored variables with all paths. In a Type 4 test, the sequence of input and stored variables must also be considered. Clearly Type 3 and 4 tests are unfeasible and only tests lying between Type 1 and 2, utilizing a sampling from the variable space, are realistic.

In the case of a Type 2 test one must still be able to enumerate the paths in the program and decide on a set of test inputs which will traverse each path at least once. This is far from a trivial problem in the case of a large program. This report presents lower and upper bounds on the number of paths in a program as a function of the number of deciders. These bounds serve as a first estimate of the work involved. A decomposition algorithm is then given which allows a graph to be cut into many smaller subgraphs. Furthermore, a second algorithm has been developed which can be programmed to machine-identify all paths of a program under test.

An implementation of a complete driver of Type 1.5, that is, of type higher than 1, is then thoroughly described, together with the algorithms to define paths and force execution. The algorithm has been implemented and a driver program designed to force testing of PL/1 programs; however, it is shown that these techniques could be extended to almost any programming language.

An example of a program run with analytical debugging techniques will also be considered and some evaluation of the usefulness of the system will eventually be given in the light of the accumulated experience.

2.3.2 Introduction

At the present state of the programming art, there exist two techniques for removing errors from a program during various stages of development, program proofs and program testing. Although much effort has been expended on program proofs, it is not clear whether this method will become a practical and widely used technique. The present universally used technique is to test to remove bugs, either by code reading, by walkthroughs, or by machine testing.

In order to investigate a strategy for testing -- be it manual, semi-automatic or automatic -- it is necessary to provide some theoretical background, such as formal definitions and analytic models, to fully define the range and scope of the test project. In general, it is indeed unclear what really is meant by error models, debugging procedures, and other such terms. We describe here a hierarchy of testing models. The importance of testing cannot be exaggerated, because only a well-tested program can be assumed to be reasonably error-free, in the prevailing lack of general techniques to prove the correctness of procedures.

Much of the testing presently done is ad hoc and heuristic rather than having any theoretical background. The purpose of this paper is to present some models and analytical techniques which can be used in developing software test systems. It will be shown, from formal definitions of testing types, that practical driver systems for automatic testing can be implemented.

The test type to be discussed in detail is a Type 1 test, which is defined as a test model in which each program path is force-traversed once. The definition involves a discussion on how program branching points and loops affect the number of paths. The process culminates in an algorithm for identifying all program paths.

The possibility of implementing and automating such a testing model is then investigated. It is shown that the technique is feasible; a system of programs has been implemented to force execution through all possible paths of a given program under test. This requires that the system analytically determine all program paths from the code, modify the input code and drive several runs of the program. Study of these forced runs will result in many program errors being caught without having to calculate and insert particular testing data; definitely, a major effort if done by hand for a complex program. The computer output for each run contains a unique labeling of the path traversed, related error messages and normal output[1], if any, and the amount of time elapsed during that run. The system has already proved very valuable in program debugging on a few problems, with its fully automatic mode of operation being the significant asset.

In the following pages we will first describe the types of tests, then the analytical determination of paths from the program flowchart. Next we will tell about driver systems and associated algorithms, and the results and limitations of the system. Finally we conclude by considering advantages and disadvantages of the model and propose future directions in the research effort.

---

[1]Note that forced testing may result in program outputs which differ from those produced by natural testing; however, these can be readily identified by the tester.

2.3.3 Types of Tests

We shall begin with a formal definition of various types of testing strategies. We shall note that, in devising a classification scheme for testing models, it is natural to desire that it correspond to an increasing (or decreasing) hierarchy of thoroughness, and difficulty. Clearly, the upper range of our numerical scheme should correspond to an exhaustive test. At the lower end of the range we will require only that each instruction be executed at least once.

We might liken the types of tests to the test procedures which an owner might apply to test a new car he has just purchased from a dealer. The first and most rudimentary check would be to compare the list of accessories ordered with the delivered list on the car window, and to see if these are present and work. For example, the owner might check to see that he got an AM/FM radio, and that it works on both AM and FM; that he received a V-6 engine and not a straight six or a V-8 and that the engine starts, that the hood lamp, glove box lamp, and trunk lamp were installed and work, etc. This checklist type test would be of the lowest test level. At the other extreme would be functional testing, for example, the use of the auto for three months. However, in between, he would try many things during his first week of driving: drive the car up a hill with and without the air conditioner on, try the heater on a cool day and the air conditioner on a hot one (or alternate the two functions), accelerate from rest to 60 mph, and try a panic stop, and so on.

Thus, the philosophy for test classification which we will use applies to product testing in general; however, the specific details will apply to software in particular.

a) Completeness and Continuity Checking - Type 0

This type of testing requires that each instruction be exercised at least once.

Intuition tells us that in testing a mechanism one basic principle is to try and exercise the parts. In the case of a program, such a test is very much expedited by a modern compiler which produces counts of the number of times each statement is executed. Type 0 test is a necessary but not sufficient condition for thorough testing of the program. In fact, when such a test is employed, one often finds design flaws. For example, it is sometimes impossible to reach a section of code, and upon detailed investigation, we find that an error was corrected by inserting a patch to bypass a block of code; however, such block was never removed and just remains inert.

We will call this lowest level type of test a Type 0 test. Obviously, it can be performed at the module level as well as at the system integration level; however, it is more common to allow the individual coder (or tester) freedom at the module stage to proceed as he wishes. Thus, much of our definition of test types is more applicable to integration testing.

b) All Paths Force Executed - Type 1

One of the problems in testing a program at a level higher than zero is the dependence between the data and the decider predicates - expressions which control the branching of an IF-THEN-ELSE or DO WHILE instruction - in the program. Intuition tells us that once we have completed a check list for a Type 0 test, we should next test all paths in the program. If we use a flowchart as our program abstraction, we can define all paths of the chart; however, it is unfeasible to find by manual analysis all possible executable paths in most programs. Thus an automated tool is highly desirable.

In the solution to the problem of constructing a program testing tool, it is convenient to define two classes of path tests: force execution-Type 1, and natural execution-Type 2. By natural execution we mean that the tester (human or machine) reads the decider predicates, computes whether they are true or false based on the current values of the program variables, and branches left or right accordingly. This concept applies as well to 3-way or multi-way branching, because such can always be expressed by 2-way branches; modern IF-THEN-ELSE constructs express indeed this fact that a condition is either true or false.

The execution time of a program is often largely devoted to the repetitive execution of DO loops within the program. However, the philosophy of a forced test is to execute all paths which only include two executions of a program loop at most. Thus, we must invent a technique to ensure that each DO loop is traversed no more than twice. We also know from experience that many errors are committed when we exit from a loop. Thus, we define the forced execution of a DO loop as testing the loop twice, once for the first execution and again for the last execution. Methods of forced execution of paths and of DO loops are discussed in Section 2.3.5.

Another question which is relevant to force traversal methodology is whether or not input data have to be supplied at execution. Obviously, data which affect the flow of control are not needed and can be omitted; however, other data types, such as operands in arithmetic expressions, may profitably be submitted if the user is interested in such testing. Therefore, it essentially depends whether the user of force-traversal testing merely wishes to check the control flow and path structure for which no input data are necessary. If he'd like to test for consistency of results, he must supply input values. At any rate, a Type 1 driver can run without any input data whatsoever; and it is this fully automatic mode of operation that renders the model so attractive. The fact that some program variables may contain meaningless quantities is but a natural limitation of this type of model and can be solved only by an escalation to a higher testing model; nevertheless, the present strategy greatly enhances the user's access to thorough testing of programs.

Based on this discussion, a force execution of paths is called a Type 1 test.

c) Exaustive Testing - Types 3 and 4

Similarly, we define here two types of exhaustive tests. A Type 3 test is an exhaustive test for a system where input sequence and initial conditions are fixed. A Type 4 test is an exhaustive test for a system where either the initial conditions, or the input sequence, can change during program execution.

In Table 1 we summarize the test class definitions which we have evolved, and discuss three typical "in between" classifications.

TYPE 0	All instructions in code executed at least once (check list).
TYPE 0.5	Many paths force executed at least once.
TYPE 1	All paths force executed at least once (simulated 100% coverage).
TYPE 1.5	All paths force executed, some naturally executed.
TYPE 2	All paths naturally executed at least once (path coverage 100%). This test is not unique.
TYPE 2.5	All paths naturally executed for many values of input parameters.
TYPE 3	All paths naturally executed for all values of input para meters (exhaustive test).
TYPE 4	All paths naturally executed for all values of input parameters, all sequences of inputs, and all combinations of initial conditions (exhaustive test for multiprocessing, multiprogramming, and real time systems with non-fixed input sequence).

TABLE 1 -- Classification of Tests

2.3.4 <u>Analytical Determination of Program Paths</u>

In this section we analyze the relationship between the number of decider predicates in a loopless program and the number of program paths.

First, an upper and lower bound are determined. Then a decomposition procedure is explained and an example is given which shows how all possible paths in a program flowchart can be identified from its structure.

a) <u>Bounds on the Number of Paths in a Loopless Program</u>

The important properties of flow charts are:

(1) the number of decision elements (deciders);

(2) the number of points where two or more feed forward branches meet (merges);

(3) the number of points where a feed forward path meets a feedback path and creates a loop.

For simplicity we assume that the graph has no loops and we bound the number of paths by the number of deciders and merges. In Fig. 1(a) we show a graph with m deciders and m merges. Each decider-merge pair furnishes two paths and by virtue of the chain structure we see that the number of paths for the total graph is simply the product of each subgraph path, i.e., 2^m paths. In Fig. 1(b) we portray a structure with n deciders and one merge. The first decider creates two paths. The next decider takes up one of the paths as its input and creates two new paths. Thus, there are n+1 paths in this graph. As an example of the application of these bounds, consider a graph with 13 deciders. The number of paths in such a graph is between 14 and 8192. Fortunately, the number of paths in a program is usually close to the lower bound.

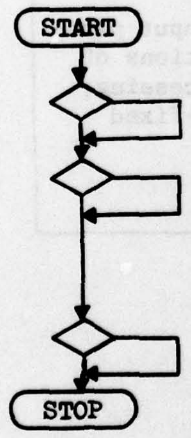


Fig. 1(a). An upper bound.

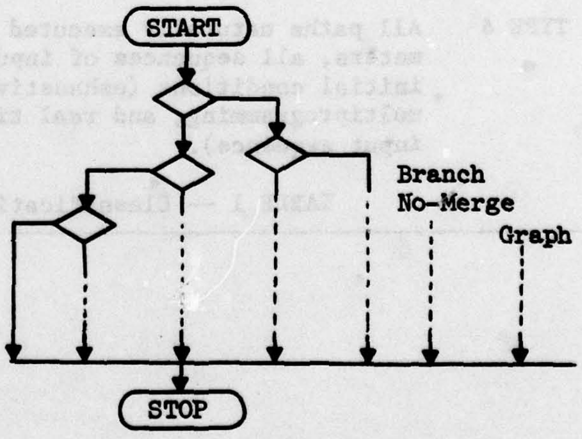


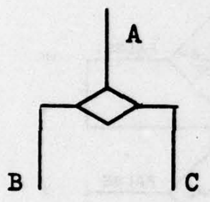
Fig. 1(b). A lower bound.

FIGURE 1. Graphs illustrating bounds on the number of paths in a flow chart.

b) <u>Manual Determination of the Number of Paths From a Flowchart</u>

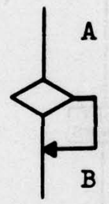
If a program is written in structured top-down form or any other modular form, the program can easily be divided into independent subgraphs. In the case of a nonstructured design, subdivision can still be performed with analogous techniques.

In performing subdivisions, the elementary substructures given in Fig. 2 are encountered. In Fig. 2(a) the number of paths in the program between point A and stop or stops is denoted by N_A. Clearly this number is the sum of the number of paths attached to the left hand branch N_B and those attached to the right hand branch N_C. In Fig. 2(b) the branch merge structure multiplies the number of paths seen at point B by 2. In the case of Fig. 2, we end up with two equalities at the merge as shown.



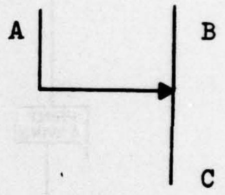
$N_A = N_B + N_C$

Fig. 2(a). Branch.



$N_A = 2N_B$

Fig. 2(b). Branch-Merge



$N_A = N_C$

$N_B = N_C$

Fig. 2(c). Merge.

FIGURE 2. Elementary graph sub-structures.

We will illustrate the calculation of the number of paths in a program with n conditional jumps. From the previous discussion, we know this number to be in the range $(n+1, 2^n)$. Let us now consider the following example:

Assume that the computer is to determine the winner of a card game in which player A is dealt two cards: A1, A2, and similarly player B is dealt two cards: B1, B2. If the players have any pairs, the highest pair wins, otherwise the player with the highest card wins. If both players have the same high card, then the winner is the player with the highest second card. Identical hands with or without pairs are ties. A flow chart for this program is given in Figure 3. There are 13 deciders, and each branch is identified with letters A, A', B, etc. The flowchart is decomposed in submodules as in Figure 2. The simple algebraic relationships which can be derived are listed in Table 2. All paths are identified and taken into account one by one; the final computation for this structure with 13 deciders yields 100 paths.

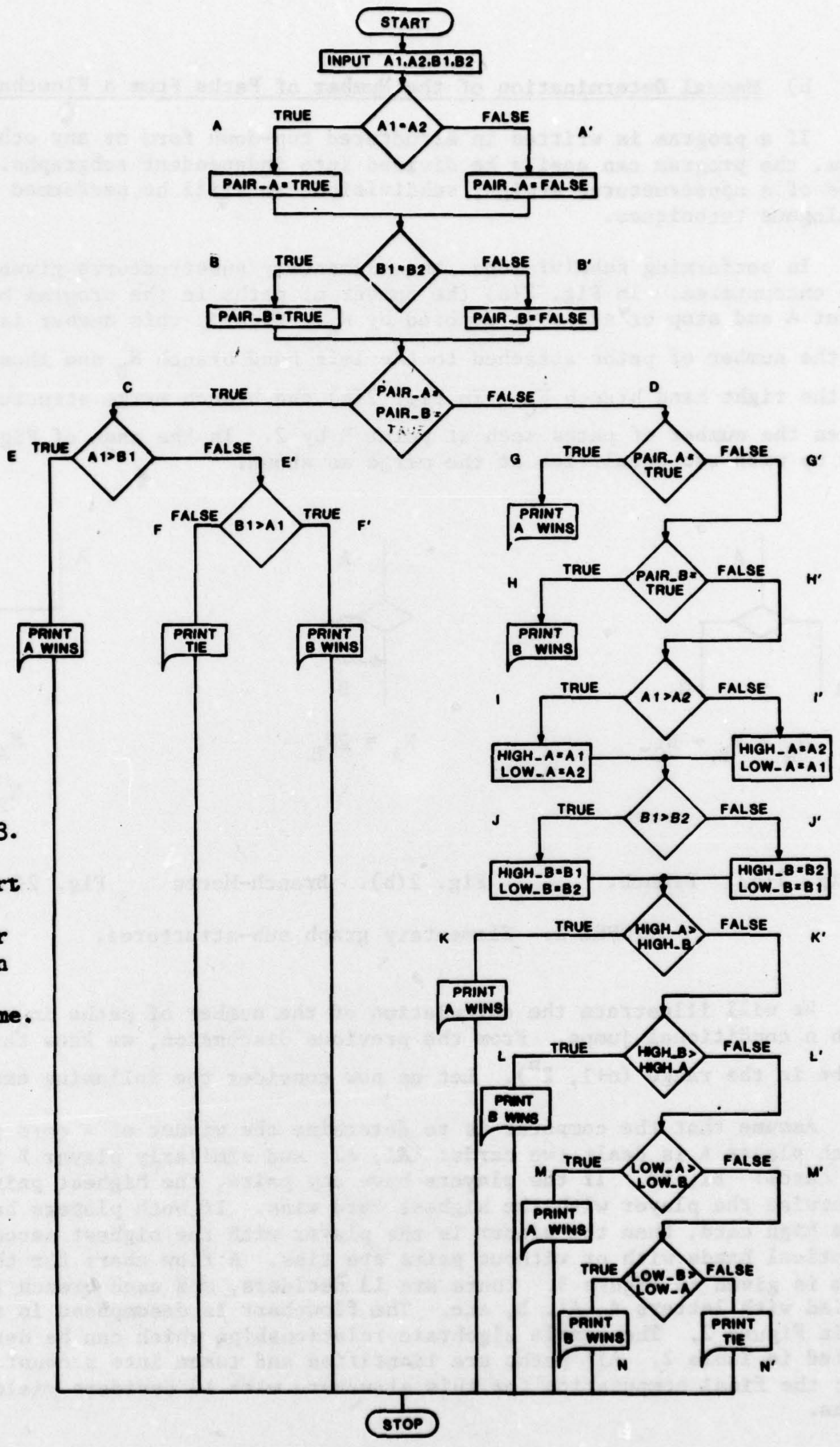


Figure 3.

Flowchart for computer solution of a card game.

Algebraic Relationship	Number of Paths
$N = N_A + N_{A'} = 2 \times N_{A'}$	$2 \times N_{A'}$
$N_{A'} = N_B + N_{B'} = 2 \times N_{B'}$	$4 \times N_{B'}$
$N_{B'} = N_C + N_D$	$4 \times (N_C + N_D)$
$N_C = N_E + N_{E'} = 1 + N_{E'}$	$4 \times ((1 + N_{E'}) + N_D)$
$N_{E'} = N_F + N_{F'} = 1 + 1 = 2$	$4 \times (3 + N_D)$
$N_D = N_G + N_{G'} = 1 + N_{G'}$	$4 \times (3 + 1 + N_{G'})$
$N_{G'} = N_H + N_{H'} = 1 + N_{H'}$	$4 \times (4 + 1 + N_{H'})$
$N_{H'} = N_I + N_{I'} = 2 \times N_{I'}$	$4 \times (5 + 2 \times N_{I'})$
$N_{I'} = N_J + N_{J'} = 2 \times N_{J'}$	$4 \times (5 + 4 \times N_{J'})$
$N_{J'} = N_K + N_{K'} = 1 + N_{K'}$	$4 \times (5 + 4 \times (1 + N_{K'}))$
$N_{K'} = N_l + N_{L'} = 1 + N_{l'}$	$4 \times (5 + 4 \times (1 + 1 + N_{L'}))$
$N_{L'} = N_m + N_{M'} = 1 + N_{M'}$	$4 \times (5 + 4 \times (2 + 1 + N_{M'}))$
$N_{M'} = N_N + N_{N'} = 1 + 1 = 2$	$4 \times (5 + 4 \times (3 + 2)) = 100$

TABLE 2. Calculation of the number of paths in the flowchart of Figure 3

2.3.5 Driver Systems

We will now introduce the practical implementation of Type 1.5 driver systems. Such drivers force the traversal of a given subject program through all its paths.

Recall that if we naturally execute a subset of all program paths, then we refer to such a test as being between Type 1 and Type 2. Similarly, in most cases forced execution will coincide with natural traversal of some paths and forced traversal of the remainder; consequently, we describe the drivers discussed here as resulting in Type 1.5 tests.

The design of drivers has evolved through several phases during the present research on testing models. The most obvious technique for complete path traversal is referred to here as an "upper bound" driver and is described in section 2.3.5(a). Such a design, as it will be shown, achieves the goal of automated testing at a high penalty. Further considerations and refinements of the problem, namely, the realization of an algorithm for path analysis, have led to the implementation of a system of programs which constitute the whole driver system. These will be described in a future technical report on this topic.

a) An "Upper Bound" Driver

The system described here was a first attempt to implement a driver to force the execution of a PL/1 program under test, from now on referred to as the subject program.

The subject program is written in standard PL/1 with no restrictions; there are only a few precautions the programmer must take in designing his code:

o	the total number of IF-statements and repetitive DO-groups, herein called NTESTS, must be supplied on a data card;	
o	each statement of the form	IF cond ... must be written as IF F(cond) ...
o	each statement of the form must be written as	DO I=limit1 TO limit2 BY increment DO I=GL(limit1, limit2) TO GH BY increment
o	each statement of the form must be written as	DO WHILE(cond) DO WHILE(H(cond))
o	functions and subroutines must be internal	

The deck of the subject program is then simply inserted within the deck of the driver program at an appropriate location. The driver exercises all paths through several runs.

The driver's mode of operation is simply based on the fact that the upper bound on the number of possible paths is 2^{NTESTS} (see section 2.3.4 a)) The driver program will internally construct a binary number, called control word, with NTESTS bits, whose initial value has all bits set to 0; this number is increased by 1 at each run during execution, until the control word has all bits set to 1.

During each of the runs, function F (as well as GL, GH and H) replaces the value of the condition with the corresponding bit from the control word. Functions GL and GH cause a DO-group with index variable to be executed once with the initial value of the index (bit=0) and once with the final value (bit=1); function H causes execution of a DO WHILE group exactly once in any case.

Since there are 2^{NTESTS} possible distinct values of the control word, there will be exactly 2^{NTESTS} runs of the subject program; therefore coverage of all possible paths is mathematically guaranteed, i.e., each path will be covered at least once. Hence the goal of automated force traversal is fully achieved with this simple strategy.

Because the number of paths in a program is usually closer to the lower bound NTESTS+1 than to the upper bound 2^{NTESTS} there will be often a large number of runs which do not represent any existing paths. For instance, the flowchart of Fig. 3 has 13 deciders but only 100 paths; hence 8092 runs are wasted with this strategy. Furthermore, since the number of runs increases exponentially with the number of predicates, the running cost of such a driver becomes prohibitive very soon, even for medium size programs.

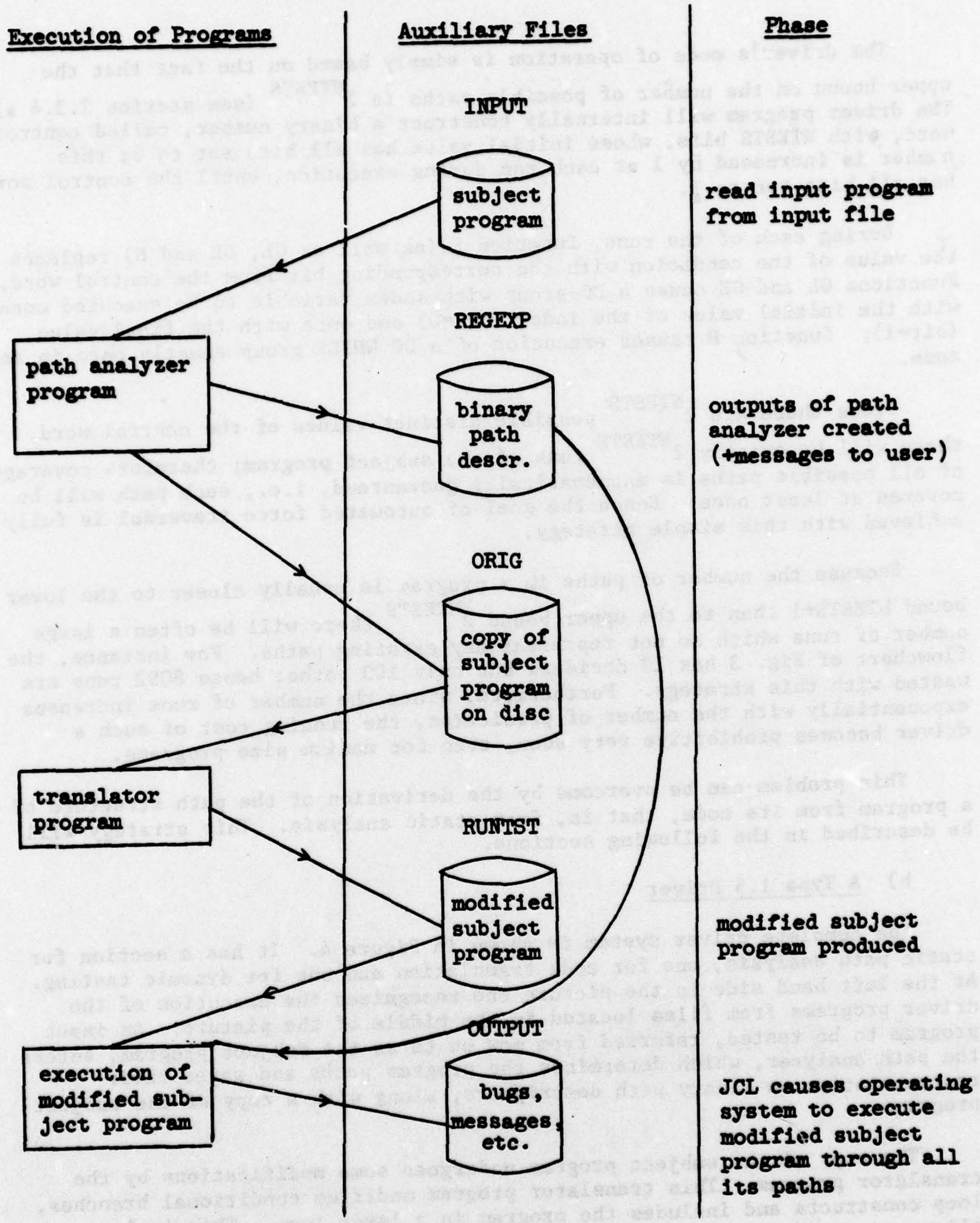
This problem can be overcome by the derivation of the path structure of a program from its code, that is, from static analysis. This strategy will be described in the following sections.

b) <u>A Type 1.5 Driver</u>

The complete driver system is shown in Figure 4. It has a section for static path analysis, one for code translation and one for dynamic testing. At the left hand side in the picture one recognizes the execution of the driver programs from files located in the middle of the picture. An input program to be tested, referred from now on to as the subject program, enters the path analyzer, which determines the program paths and saves their representation as binary path descriptors, along with a copy of the subject program.

The copy of the subject program undergoes some modifications by the translator program. This translator program modifies conditional branches, loop constructs and includes the program in a large loop. This inclusion allows repeated executions.

Eventually the modified subject program reaches the execution stage through all its paths, as determined by the binary path descriptors, and the output of the driver is produced.



Execution of Programs
Auxiliary Files
Phase
INPUT
subject program
read input program from input file
REGEXP
path analyzer program
binary path descr.
outputs of path analyzer created (+messages to user)
ORIG
copy of subject program on disc
translator program
RUNTST
modified subject program
modified subject program produced
OUTPUT
execution of modified subject program
bugs, messages, etc.
JCL causes operating system to execute modified subject program through all its paths

FIGURE 4. The system of the driver programs and their operation.

c) ***The Algorithm for Path Analysis***

Labeling of Paths: We shall use the convention to label the "true" branch of a conditional statement with a "1", and the "false" branch with a "0", as seen in Figure 5. In this way, it is possible to uniquely label a path in a given program with a binary path description, as shown in Figure 6.

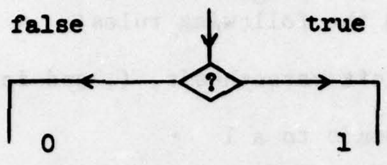


FIGURE 5. Labeling of branches.

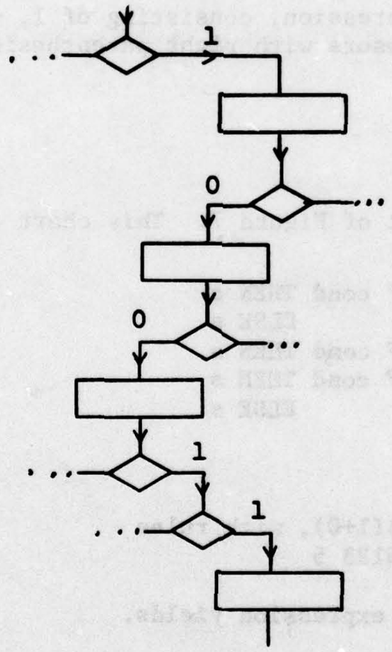


FIGURE 6. Labeling of path 10011.

Algorithm for Finding All Possible Paths: We will first show that it is possible to determine all possible paths. We will start with the path analysis for a program without any repetitive DO constructs. Since each path is uniquely defined by a binary integer, called here path descriptor, the problem of finding each path in a given program is analogous to the problem of finding the set of binary integers associated with the path structure of that program. Because sets of binary quantities can be expressed by regular expressions, we propose an algorithm which constructs a regular expression whose associated set contains the values of control binary words corresponding to paths. Only the operation + (expressing union in the associated set) and concatenation will be needed. The expression is recursively defined as being always binary, i.e., it contains two terms separated by +; a term is the symbol 1, or 0, or an expression; concatenation of expressions forms an expression.

The algorithm scans a PL/1 program in search of IF-THEN-ELSE constructs, and operates accordingly to the following rules:

1) each IF opens a left parenthesis, (, and initiates an expression

2) each THEN corresponds to a 1

3) each ELSE corresponds to a 0, and since it matches a previous THEN, a + is inserted at that level

4) if no matching ELSE is present, it is assumed to be there and +0 is therefore added

5) each balanced expression, consisting of 1, +, 0, closed at its level, causes closure with right parenthesis at that level.

2.3.6 Examples

EXAMPLE 1.

Consider the flowchart of Figure 7. This chart is implemented by the program segment

```
IF cond THEN s
        ELSE s
IF cond THEN s
IF cond THEN s
        ELSE s
```

The algorithm constructs:

```
(1+0)(1+0)(1+0), with rules
12 3512 45123 5
```

Computation of the regular expression yields.

111,011,101,001,110,010,100,000

i.e., the eight possible paths.

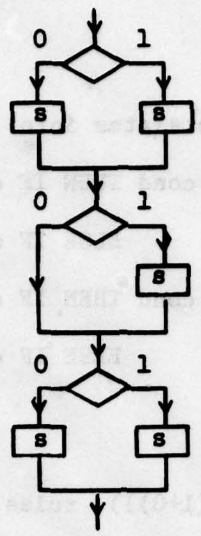


FIGURE 7. Flowchart for Example 1.

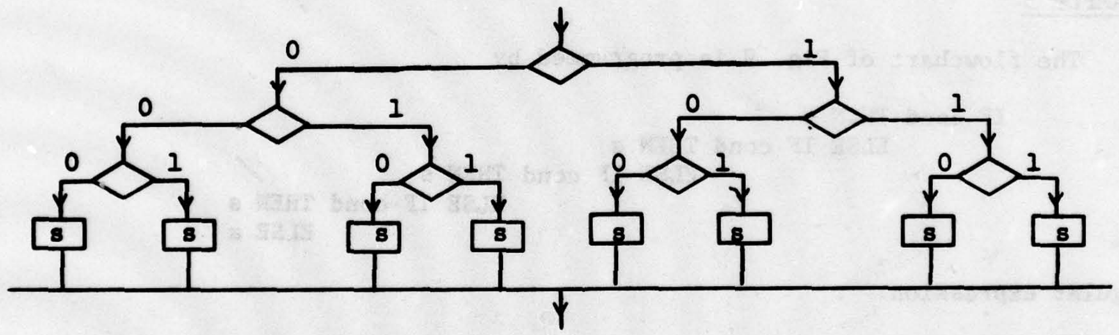


FIGURE 8. Flowchart for Example 2.

EXAMPLE 2.

The flowchart of Figure 8 translates into:

```
                    IF cond THEN IF cond THEN IF cond THEN s
                                                      ELSE s
                                          ELSE IF cond THEN s
                                                      ELSE s
                            ELSE IF cond THEN IF cond THEN s
                                                      ELSE s
                                          ELSE IF cond THEN s
                                                      ELSE s
```

Regular expressions:

(1(1(1+0)+0(1+0))+0(1(1+0)+0(1+0))), rules:

121212 35 312 35 31212 35 312 35

expressing the eight paths

111,110,101,100,011,010,001,000

EXAMPLE 3

The flowchart of Fig. 9 is programmed by

```
IF cond THEN s
        ELSE IF cond THEN s
                     ELSE IF cond THEN s
                                  ELSE IF cond THEN s
                                               ELSE s
```

Regular expression:

(1+0(1+0(1+0(1+0)))), rules:
12 312 312 312 35

which gives

1,01,001,0001,0000

EXAMPLE 4

The flowchart of Fig. 10 is realized by

```
IF cond THEN DO;
     IF cond THEN IF cond THEN s
                           ELSE s
              ELSE IF cond THEN s
                           ELSE s
     IF cond THEN s
     END;
        ELSE DO;
     IF cond THEN;
              ELSE s
     END;
```

Regular expression:

(1(1(1+0)+0(1+0))(1+0)+0(1+0), rules:
121212 35 312 35 12 45 312 35

yielding

1111,1110,1101,1100,1011,1010,1001,1000,01,00

In Example 4, the path structure is complicated by the presence of DO groups within the THEN and ELSE clauses. When this occurs, construction of the regular expression temporarily halts, and the algorithm calls itself recursively for that DO module.

This algorithm has been tested with a program written in LISP which was then translated into a PL/1 program.

A second algorithm solves the regular expression and finds all the elements of its associated set; this number is the number of all possible paths, hence the algorithm, as an extra bonus, enumerates all paths of a program.

The complete analyzing program embodies these algorithms. It starts by reading in the subject program, card by card. This is stored as a character string in main memory and is saved on an external file, called ORIG. Control is then passed to the section performing the scanning and computing the set of binary control words.

The scanning mechanism is built around two PL/1 procedures, CAR and CDR, which respectively return the head and the tail of a given string. By <u>head</u> we mean: any PL/1 operator (such as *,**,— ,), any PL/1 separating character (such as ;, :, (),)or any string separated by an operator, by a separating character or by blank, or any quoted string of comment; by <u>tail</u> we mean the string without its head. Hence CAR is capable of correctly picking up keywords in portions of statements such as ;IF(,$_*$/THEN/$_*$, L2:DO;, but will not pick them up in THEN1=5, 'A STRING IF NEEDED',/$_*$THEN A COMMENT$_*$/.

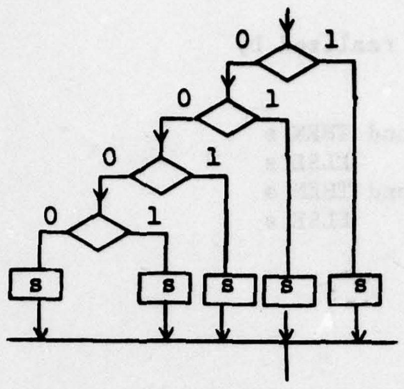


FIGURE 9. Flowchart for Example 3.

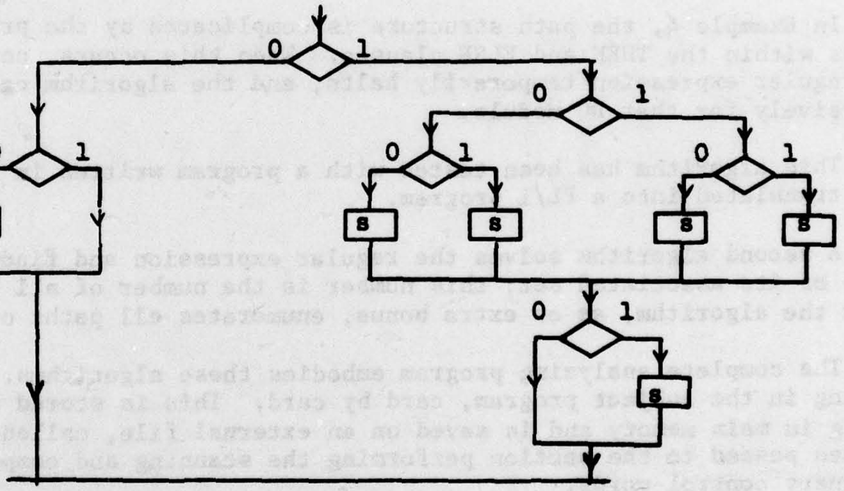


FIGURE 10. Flowchart for Example 4.

The analyzing program produces the set of binary path descriptors, saved on external file REGEXP, which will control the test portion.

2.3.7 Results

In this section we shall discuss the usefulness of Type 1.5 drivers for forced testing, comparing their use with other widely used testing strategies, and describe the efficiency and deficiencies of the present implementation.

a) Automatic Analysis and Forced Execution

We have shown in Section 2.3.4(b) that even in a simple program like the one of Fig. 3, path identification and enumeration is not a trivial problem, but time consuming and error prone. In such cases, the algorithm described in Section 2.3.5(c) has great advantages, because it not only identifies all paths and properly labels them, but the path analyzer program saves all binary path descriptors. Hence it is not necessary to spend time in designing data sets to exercise all paths (with the result of not covering them all, as it generally happens). The driver system takes the input program; the analyzer creates the descriptor for each run or path, the translator modifies the code, and the modified program runs 100 times through all its paths.

b) Comparison Between Manual and Automatic Testing

We established a timetable to compare results and time spent by testing with input data and with the driver system. We began by saying that it took 30 minutes to design a successfully compiled program. We then tested the program manually and automatically.*

Manual Testing with Data	*Automatic Testing with the Driver*
● design of a data set to exercise some paths: 10 minutes	● path analyzer: 1.13 min,, 450k core.
● running of the program through these paths: 0.01 minutes	● translator: 0.27 minutes
	● translated subject: all paths
● results: program is OK	● results: one bug was discovered

(The bug appeared in the form of two contradictory outputs; an erroneous ELSE clause was subsequently fixed in the subject program.)

It appears therefore that no particular penalty in running time has to be paid to let a program run through all its paths. Recall in fact that in general the number of paths will be closer to the lower bound than to the upper bound. Moreover, program with repetitive DO-loops may very well run faster through the driver than with some manual testing, because extra saving in running is achieved by not letting loops run more than twice. This saving may become very consistent in a program with many nested loops.

* These operations were carried out on the IBM 360/65 computer at the Polytechnic Institute of New York, Brooklyn, NY 11201.

Since it is not necessary to design any testing data, the programmer is saved time and effort for manual debugging. There is, moreover, no guarantee that testing with data achieves the result, as this case shows the erroneous path was accidentally never reached with testing data.

The path analyzer program, however, is not very efficient at present and requires prohibitive amounts of computer memory, as discussed in the next section.

c) Efficiency of the Package

The path analyzer program is a literal implementation of the recursive algorithm described in Section 2.3.5(c). As such, it is slow and inefficient and requires a large amount of computer memory. This is due to the fact that each time a recursive procedure which is still open calls itself either directly or indirectly, it reallocates a new space in the main storage. In PL/1, unfortunately, a string of maximum length is allocated even if the string is declared as variable length. As a result, the path analyzer wastes a large number of bytes. This wasted memory is recuperated only when the last recursive call terminates, at which point the recursion stack begins to pop.

The problem has in fact been avoided by implementing the algorithm for path analysis in LISP, a language ideally suited for recursive function and string (i.e., list) manipulations. No prohibitive memory is needed, since the pairing by pointer mechanism uses space only when needed. However, the LISP interpreter is not widely available, and is in general very slow compared to compilers for other high level languages.

One solution to the inefficiency of the analyzer we are considering would be to still keep the recursive algorithm in PL/1. However, instead of storing in the stack the character string resulting from an intermediate computation, we would operate always only on one copy of the string and save rather the present scanning location, i.e., a one-word pointer. Hence we would achieve a saving of 32K-bytes versus one word, which is close to an order of magnitude of 4. A drastic saving in memory space would be realized, and simultaneously some techniques to speed up the algorithm could be added.

The translator program is essentially built around string manipulating built-in procedures, and is therefore fast. No doubt, however, it could still be further optimized.

The translated subject program will run through all its paths. In the previous section we point out that this may be achieved at the price of a high penalty in running time. It is worth mentioning, however, that even if this were so, the advantage provided by an automatic mechanism which is guaranteed to exercise all paths in a subject program outweights some possible disadvantages of extra running time.

d) <u>Present Restrictions on the Subject Program and Limitations of the Driver System</u>

Our system implements a driver system in order to force execution of a subject program written in PL/1 through all its possible paths. Almost all PL/1 can be used in the design of the subject program, with a few restrictions. Some depend on our computer system and some were introduced for ease of design of the system. Some are tolerable or even welcome when they encourage good programming style. Some are undesirable and will hopefully be removed in the next future.

As shown in Section 2.3.3(b), the driver will force execution of paths that could never be reached by natural execution, hence capturing non-existent errors. This is a flaw of a Type 1 testing model and could conceivably only be solved by a higher model. It is however, unclear whether the determination of impossible paths can be solved at all. Assume that within certain lines of code, an algorithm is called for the computation of one of the values used in the next decider. At testing, it is unknown whether the algorithm will terminate or not (i.e., the halting problem). This example shows that it is in general impossible to exercise a path, even at level 1. Our driver, in fact, discontinues a path if it takes longer than a certain time, thus putting a bound in order to avoid solution to the halting problem. Possibly, only those impossible paths depending on assignments of known values to the predicate variables can be identified, and possibly only a subset of them could be found by static analysis. In the meantime, therefore, we shall regard this problem as not highly important; the user can in fact quickly identify these paths from the driver's output.

We will now consider a few implementation restrictions. Our version of PL/1 does not allow strings, that is, subject programs with more than 32767 bytes (about 400 cards). Without modifying our local definition of PL/1, this could again be solved by a swapping mechanism, which would call from disk the different sections of a program under analysis or translation.

Ease in program design of the analyzer requires the exclusion, in the subject program, of any variable or identifier named IF, THEN, ELSE, DO, END, BEGIN, TO, BY and WHILE. Similarly, multiple clause DO-groups (clauses are separated by commas), and multiple closure with a labeled END -- all this is allowed in PL/1 -- have to be avoided. However, use of these PL/1 features often causes confusion, and often programmers are independently encouraged not to use them.

Moreover, although it is a minor drawback that the driver cannot handle GO TO's -- they can always be avoided in structured programs, and almost always with profit -- it is more serious that it cannot cope, at this stage of development, with branching to subroutines. A mechanism could be added in order to allow scanning to jump to the subroutine location, saving the current scanning location, to which to return upon exit from the procedure. This is conceptually simple but will involve some tedious readjusting of the driver's code; however, we plan to do this in the near future.

It is unclear, however, how a static path analyzer could cope with recursive procedures, because the path structure of a recursive program is unknown before execution time -- in fact, no graph or flowchart representation exists for a recursive algorithm. Let us just mention the fact that any recursive function can always, though sometimes painfully, be made non-recursive; besides, recursive programs are rarely found in the real world of programming.

Except for its inability to handle procedures, the driver system is therefore general enough to accept any well written PL/1 program to be tested.

2.3.8 Summary and Conclusions

We conclude by describing a practical application for this implementation of a Type 1.5 testing model.

The described driver system can be integrated into an operating system. A program would initially be compiled to catch syntax errors and upon successful compilation, it would be submitted to the driver system for forced execution. Therefore, another set of errors, appearing at execution time, could be eliminated prior to testing with real data. Alternatively, a strategy intermingling natural and forced execution could be implemented.

We would like to recall once more that our effort, although directed mainly toward PL/1 programs, can be extended to other programming languages as well. We hope therefore that our driver techniques represent a step toward automatic debugging.

As a final conclusion, we notice that:

1. Although the area of testing is a difficult one, we have developed several quantitative models and approaches to aid research progress in the field.

2. A quantitative way of describing and categorizing different types of tests has been developed which may aid discussion and characterization of tests.

3. A system of algorithms to perform automatic Type 1.5 testing has been implemented and described in detail. It was shown that such a driver is indeed feasible and advantageous. An explicit computation of the number of paths in a graph was carried out analytically. The same example has been submitted to the driver system and run through the driver. This illustration can be extended to other examples and generalized. Hence, the testing model has been defined, researched and fully implemented. Although the model has already proved itself a useful tool, it is hoped that it will clarify and further stimulate research in this area.

2.4 STATISTICAL ASPECTS OF PROGRAM TESTING by A.E. Laemmel

In a previous report (3) several formulas were derived which relate the probability of program error to the number and type of tests previously made on that program. No program of any size can be expected to be perfect, if for no other reason than the difficulty of defining perfection. Thus, if "error" is suitably defined, and if only a limited number of tests can be made on the program, there will be some finite probability of error. This progress summary will not discuss the general validity of applying statistical theory to the area of programming, although this topic is important. Rather, the interpretation of "statistical dependence between errors" and "program bugs" will be examined.

The central result of the previous report was (see Eq. 15, ibid.):

$$P_e = \prod_{i=1}^{\tau} (1-\beta_i) \sum_{j=t+1}^{N} q_j \left[1 - \prod_{i=\tau+1}^{M} (1-\sigma_{ji}\beta_i) \right] \tag{1}$$

where	P_e	= probability of tester acceptance and user error
	N	= number of input values
	t	= number of input values tested
	M	= number of possible bugs
	q_i	= probability of user selecting i th input value
	β_i	= probability of the i th bug
	σ_{ij}	= 1 if the j th bug causes an error for input i
		0 if the j th bug doesn't effect input i
	τ	= number of bugs causing at least one error.

We assume that the matrix σ has already been permuted so that the tested values (first t columns) and related error-causing bugs (top rows) come first. This is for convenience, and causes no loss of generality.

It is important for us to define what we mean by a program bug. Suppose that a program has N different input values, and that failure to give the correct output for one input value implies nothing at all about possible failure for another input value. If this situation really existed, it would require an impossibly large number of tests to insure a reasonably small value of P_e. Note that throughout this discussion, the concept "one test" means running the program for one value of its input data, not testing one path through the program, nor testing one module of the program. To this end the matrix σ has been introduced so as to tie together these two views of testing. Formally, the introduction of the matrix σ might be said to describe the statistical dependence among errors for different input values. It is implicit that the program as a whole must be tested, i.e., separate modules cannot be individually tested in this model. Several possible ways to define a bug will now be defined and compared:

i) The program has M modules and bug i means that the i^{th} module doesn't work properly. This will cause errors only for those input values which cause the program to call the i^{th} module.

ii) The program has only one module with one input variable, and, for example, bug 1 means it fails for all positive input values, but 2 means it fails for 0 input, and bug 3 means it fails for all negative input values.

iii) The program has M lines, and bug i means that the i^{th} line has an error.

iv) Some combination of the above. For example, bug 1 might refer to negative input values, bug 2 to subroutine B, etc.

These and other possible interpretations must be compared by considering how realistically they correspond to actual programming situations, and to the relative difficulty of gathering data and statistics which can be inserted into the formula (1). To start with, (iii) seems attractive since all programs are line or statement structured. Perhaps one might like to consider that each line might have n possible bugs, so that $M = nL$. The big difficulty with (iii) is that the error might be a <u>missing</u> line, say a variable that was not initialized. If one allows for m possible missing lines, then $M = nL + m$; but the effect of a missing and unknown line on a particular input value is hard to anticipate. Somewhat the same objections can be made to (i). In particular, many additional possible bugs exist as incorrect linkages between modules. While (ii) may appear to avoid the difficulties just mentioned, it requires more knowledge of what the true outputs should be for all input combinations. In other words, (ii) is problem oriented instead of being program oriented, and this might make the gathering (or simply guessing) of statistics more difficult.

In addition to properly interpretting the bugs, Eq. (1) above should be simplified if it is to be used in practice. In the previous report, (ibid, Eq. 16) a simplified version was given, but this is perhaps too simple. Work is going on in finding a useful compromise between these extremes.

2.5 COMPLEXITY MEASURES by H. Ruston, A.E. Laemmel, and E. Berlinger

A new complexity measure A preliminary report on complexity measures is partly completed. The report first reviews and contrasts several of the measures introduced so far. It includes: (1) the measure based upon division of inputs into classes, with each input class resulting in a single output (Hellerman); (2) the module vector measures (McTap, Chapin), applied to deciders and loops exceeding heuristically assigned reference values; (3) the classification measures (Sullivan); (4) the topological measures (Chen, McCabe, Myers); (5) the measures based upon length (Halstead, Shooman, Laemmel).

In the second part a new measure based upon information theory is introduced. This measure is based on those aspects of programing which present difficulty to the program. It is assumed that the less frequently an item is used, the more difficult it is for the programmer to use the item correctly.

Let p_i represent the probability of usage of the i^{th} type in a programming language, and f_i be its frequency of usage in a module or program. Then we can define a complexity measure for that module as

$$C = -\sum f_i \log_2 p_i$$

The p_i's are the long term probabilities and can be measured over a period of weeks, months, or even years, and can be constantly or intermittently updated using automated techniques. The measure c itself can be automatically obtained.

If C_j is the complexity of the j^{th} module, then we can define the program complexity as $C = \sum_j C_j$.

A number of different schemes can be used to calculate this measure. For instance, all operations and operands could be counted. Operations, such as CALLs, which include names unique to a program can be ordered according to their frequency of usage. The module called with highest frequency would be paired with the highest probability. A similar technique would be used with operands.

Another technique might be to group certain operators together and combine their probabilities. As an illustration of such grouping, there does not seem to be any reason to assume that addition is either more or less complex than subtraction. Thus addition and subtraction may be treated as one operator as far as the complexity measure is concerned.

We may wish to distinguish among levels of nestings. A DO statement at the second level may be more complex than one at the first level. The higher nestings could be assigned separate frequencies and have individual probabilities.

There are a number of advantageous features in this measure. Since the measure is based on the probability of usage of each type, the measure is sensitive to the natural tendency of humans to forget what they do not use. The measure can also be made sensitive to the different levels of nestings. No other measure exhibits such sensitivities. Furthermore, once the necessary programs are in effect at a particular installation, the calculation of the frequencies, probabilities and the ensuing measure can be completely automated.

Program complexity and information content. The purpose of a computer program is to convey information from the user to the computer-operating system. The information is that needed to specify the desired mapping of input data to output data. Since the operating system and the computer hardware usually contain many commonly used functions and programs, the user's program need not contain complete details of the desired computation. The amount of information contained in a program can be calculated by several methods, and it might well be identified with program complexity.

Ever since Shannon's paper of 1948 information content of a message has been regarded primarily as a statistical quantity. Suppose a user, or a group of statistically similar users, feeds to computer a series of programs $\sigma_1 \sigma_2 \ldots \sigma_t \ldots$ These programs are selected from a set of possible programs $\{\pi_1, \pi_2 \ldots \pi_M\}$. The information content of each program is at least as large as

$$H_o = \sum_{i=1}^{M} P_r(\pi_i) \log_2 \frac{1}{P_r(\pi_i)}$$

The quantity H_o will be of reasonable size, but M is astronomical and hence there seems to be no reasonable way to calculate H_o directly. There might be indirect ways to estimate H_o, and in any event it is useful conceptally.

Another way to calculate the information content of a program is to examine the statistical properties of parts of the program such as characters, operators, operands etc. Call these smaller units tokens as shown in Fig. 11,

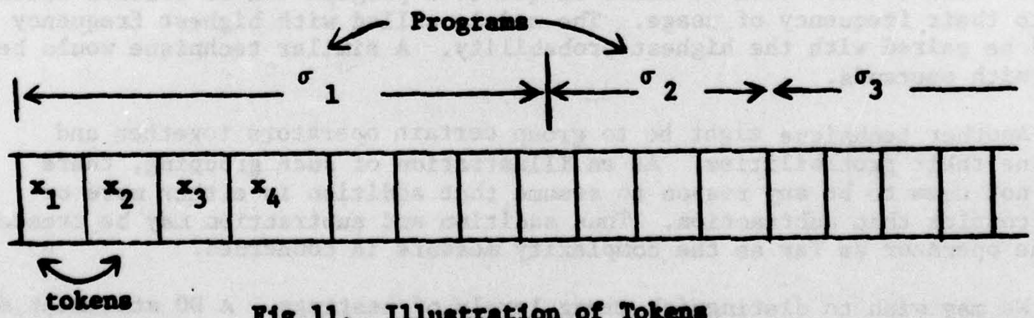


Fig 11. Illustration of Tokens

and consider the sequence of tokens going from the user to the computer. Define $P(i \mid B_j^{(n-1)})$ as the probability of the ith token following a given block of $B_j^{(n-1)}$ tokens. Define

$$F_n = \sum_{j=1}^{A^{n-1}} P_r B_j^{(n-1)} \sum_{i=1}^{A} p(i \mid B_j^{(n-1)}) \log_2 \frac{1}{p(i \mid B^{(n-1)})}$$

where A is the number of types from which each token may be selected. Since F_n is monotonic non-increasing it must approach a limit (possibly 0) as $n \to \infty$. Define F_∞ as this limit, that is

$$F_\infty = \lim_{n \to \infty} F_n$$

Let L_{av} be the average number of tokens in a program. It can be shown that

$$H_o \leq L_{av} F_\infty$$

A value for $L_{av} F_\infty$ exceeding H_o indicates that the program contains more information than actually required. This additional information might be due to comments, mnemonic labels and variable names, alternate algorithms solving the same problem, different ways to order program blocks, different ways to allocate memory etc.

Operators and operands often alternate in a program, either because the program is written that way, or because operator and operand are so defined as to force the alternation. In either case, a good choice for the token in computing F_n is a operator - operand pair. Since A is a large number, the actual calculations are greatly facilitated if the probabilities follow a simple distribution formula such as Zipf's law. Work is in progress on developing the relations between the various information measures with the theory of Zipf's law applied to operation - operand in evaluating numerical examples. This work will be reported in a future report.

2.6 MODELS FOR THE MANAGEMENT OF SOFTWARE -- M.L. Shooman and A. Kershenbaum

Initial work on two models for the management of software were undertaken. The first model considers the development time, T, to be the sum of programming time T_p and the communication time T_c, that is,

$$T = T_p + T_c$$

For N_p programmers interfacing with each other, the total number of interfaces is

$$\binom{N_p}{2} = \frac{N_p(N_p - 1)}{2}$$

If S is the total number of instructions to be developed, and T_p is measured in months, then the productivity P measured in instructions per month is

$$P = \frac{S}{N_p T_p}$$

One assumption is that the communication with each interface requires a fraction K of T, we obtain for the communication time T_c

$$T_c = \frac{KTN_p(N_p - 1)}{2}$$

Substitution gives

$$T = \frac{T_p}{1 - \frac{KN_p(N_p - 1)}{2}}$$

or

$$N_pT = \frac{S/P}{1 - \frac{KN_p(N_p - 1)}{2}}$$

If on the other hand there is a fixed coordination time for each interface, defined as

$$K_2 = \frac{\text{months coordination time}}{\text{program interface}}$$

then man-months N_pT become

$$N_pT = \frac{S}{P} + K_2N_p^2(N_p - 1)$$

The second model treats the question of decomposition of a problem into subproblems (solved by distinct individuals) and the attendant communication problems. The objective is to study the effect of the shape of the system on its overall cost or complexity. As an example, the decompositions of P into case 1 and case 2 are shown in Fig 12. In case 1 communication among subproblems, (i.e. the individuals solving these subproblems), takes place directly. In case 2 all communication takes place through a central point P_2.

A quantitative comparison of these and other more elaborate decompositions (involving more subprograms) are being studied. The results will be disseminated in a separate technical report.

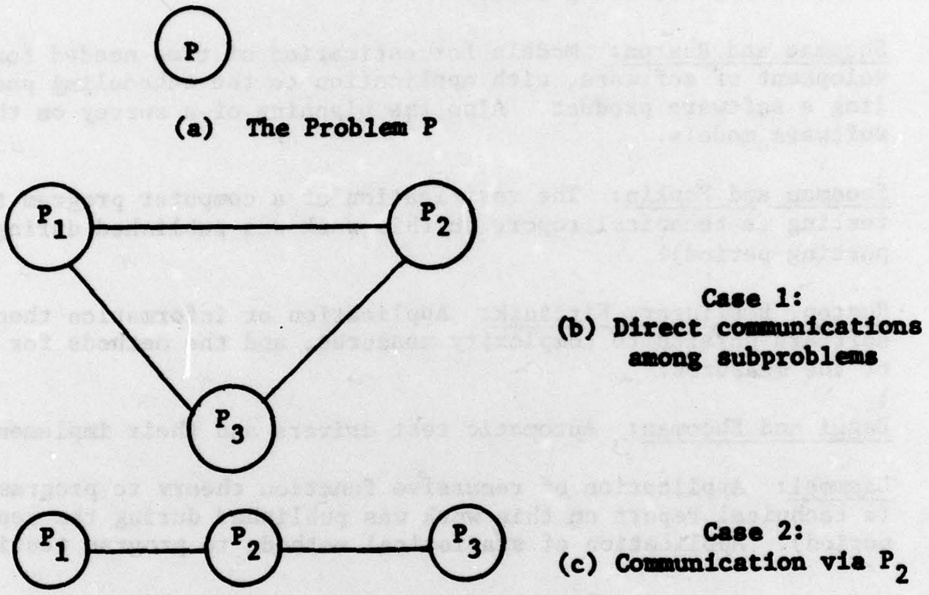

P
(a) The Problem P
P_1
P_2
P_3
Case 1:
(b) Direct communications among subproblems
P_1
P_2
P_3
Case 2:
(c) Communication via P_2

Fig. 12 The Problem P and its Decomposition

3.0 <u>REFERENCES</u>

(1) M. Lipow, "Application of Algebraic methods to Computer Program Analysis," Report TRW-55-73-10, TRW Software Series, May 1973.

(2) R.S. Garfinkel and G.L. Newhauser, "Integer Programming," John Wiley and Sons, New York 1972, pp. 302-303.

(3) M. Shooman and H. Ruston, "Summary of Technical Progress, Software Modeling Studies," RADC-TR-76-143 Technical Report, May 1976, pp. 19-30, A025895.

4.0 PERSONNEL AND WORK AREAS

The following personnel participated in the research activities during this reporting period:

M.L. Shooman
H. Ruston

D. Baggi	A. Laemmel
E. Berlinger	R. Lipshitz
J. Kitainik	G. Popkin

and worked in the following areas:

1. Shooman and Ruston: Models for estimation of time needed for the development of software, with application to the scheduling and controlling a software product. Also the planning of a survey on the use of software models.

2. Shooman and Popkin: The verification of a computer program through testing (a technical report on this work was published during the reporting period).

3. Ruston, Berlinger, Kitainik: Application of information theory and software physics to complexity measures, and the methods for evaluation of the measures.

4. Baggi and Shooman: Automatic test drivers and their implementation.

5. Laemmel: Application of recursive function theory to program complexity (a technical report on this work was published during the reporting period). Application of statistical methods to program testing.

5.0 DIRECTION FOR NEXT PERIOD'S WORK

1. Baggi -- Completion of a report on software test models and the implementation of test drivers for such models. Further work will focus on increasing the efficiency and usefulness of the path analysis algorithm and of the Type 1.5 driver system.

2. Laemmel -- preparation of a report on statistical program testing, taking into account errors not found as well as errors caused by improper correction.

3. Ruston and Berlinger -- Application of information theory to complexity measures and the evaluation of such measures.

4. Ruston and Kitainik -- Application of graph theory to complexity measures.

5. Ruston and Mohanty -- Simplification of the technique for test data selection. The present technique is unyielding and very time-consuming.

6. Shooman and Popkin -- Further work on enumeration, classification, and characterization of tests and their attributes. An effort will be made to correlate practical tests now in use with the theoretical schemes which have been established.

7. Shooman, Ruston and Cormier -- Application of Markov processes to estimation of actual time needed to development of software (taking the returns to previous stages into account). Such estimation will be useful for scheduling and controlling a software project.

8. Shooman and Lipshitz -- Further work on generalizing an early effort on modular and automatic programming techniques.

6.0 PROFESSIONAL ACTIVITIES

This section summarizes the professional activities of the research personnel working on this contract.

6.1 PUBLISHED AND SUBMITTED PAPERS AND REPORTS

1. C.L. Hsu, L. Shaw, S.G. Tyan, "Reliability Applications of Multivariate Exponential Distributions," POLY-EE-77-0361/EER122, September 1, 1977 (appeared in early 1978).

2. M.L. Shooman and H. Ruston, "Software Modeling Studies-Summary of Technical Progress," Final Technical Report, RADC-TR-78-4, Vol. I (of two), Jan. 1978, A052615.

3. D.L. Baggi and M.L. Shooman, "An Automatic Driver for Pseudo-Exhaustive Software Testing," Proceedings of the 1978 IEEE Spring Compcon Conference, San Francisco, February 28-March 3, 1978.

4. D.L. Baggi, "Implementation of a Channel Vocodder Synthesizer Using a Fast, Time-Multiplexed Digital Filter," Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Tulsa, OK, April 19-12, 1978.

5. A. Laemmel, "Study of Recursive Function Theory and Its Application to Program Complexity," POLY-EE/EP-77-037/SMART 108, May, 1978.

6. G.S. Popkin and M.L. Shooman, "On the Number of Tests Necessary to Verify a Computer Program," Poly-EE-78-047/SRS 113, June 1978.

6.2 TALKS AND SEMINARS

1. H. Ruston, Seminars on Structural Design and Structured Programming, Cherry Hill, NJ, January 5, 1978, N.Y. City April 26-27, 1978, Toronto, Canada, May 9, 1978.

2. M.L. Shooman, Talk on Software Reliability System, IBM System Research Institute, March 7, 1978.

3. M.L. Shooman, "Summary of Modeling Progress," RADC Contract Report, Griffiss AFB, NY, March 10, 1978.

4. M.L. Shooman, 5-Day Course on Reliability, Availability and Safety, Stat-A-Matrix Institute, Princeton University, April 24-28, November 6-10, 1978.

5. M.L. Shooman, "Software Engineering Models," Computer Science Seminar, Queens College, NY, May 19, 1978.

6. M.L. Shooman, "Software Reliability Models," Spring Lecture Series, N.Y. Section ACM, May 18, 25, 1978.

7. M.L. Shooman, "Future of Computers," High School Science Day, Hofstra University, Hemptstead, NY, June 2, 1978.

8. D.L. Baggi, "A Test Driver for Software Testing," Seminar, PINY, Farmingdale, NY, July 6, 1978.

9. M.L. Shooman, Series of four lectures and four discussion sessions on Software Reliability, Advanced Course on Reliable Computing Systems, University of Newcastle, England, July 31-August 4, 1978.

10. H. Ruston and M.L. Shooman, "Software Models -- Some Applications," Syracuse/RADC/IEEE Workshop, Minnowbrook Workshop, September 20, 1978.

11. M.L. Shooman, "Software Reliability Models," Martin Marietta Labs., Baltimore, MD, September 28, 1978.

12. M.L. Shooman, "Reliability Analysis," Polytechnic Graduate Seminar Series in Electrical Engineering, October 5, 1978.

13. M.L. Shooman, "Software Engineering Models," Computer Science Lecture Series. Lecture Talk and discussion leader at graduate seminar, University of Maryland, October 23, 1978.

14. E. Lipschitz, "Automatic Programming," Computer Science Seminar, Polytechnic Institute of New York, November 8, 1978.

15. M.L. Shooman, "Engineering Reliability," Polytechnic Executive Management Seminar, Skytop, PA, November 13, 1978.

16. M. Ruston, "Software Models," Computer Science Seminar, Polytechnic Institute of New York, December 13, 1978.

6.3 SYMPOSIA, TECHNICAL SOCIETIES, AND EDUCATIONAL ACTIVITIES

1. M.L. Shooman organized a group of five interrelated talks on Reliability Application in Mechanical Engineering for the Polytechnic Graduate ME Seminar, Fall, 1979.

2. H. Ruston reorganized three core courses in the undergraduate computer science curriculum and developed a new course in data structures.

3. M.L. Shooman developed new notes and material on Software Design to his graduate Software Engineering course.

4. A. Laemmel developed two new undergraduate microprocessor laboratory courses and a graduate microprocessor course.

5. M.L. Shooman serves on the Executive Committee of the Computer Society's Technical Committee on Software Engineering.

6. Various research members (Shooman, Ruston, Laemmel, Baggi, Popkin) regularly serve as referees for Compcon, Compsac, IEEE Proceedings on Software Engineering, and Networks, conferences and journals.

6.4 BOOKS AND AWARDS

1. Henry Ruston published the textbook "Programming with Pl/I," McGraw-Hill Book Co., NY, 1978.

2. Martin L. Shooman contributed a chapter entitled Software Engineering to the book "Computing System Reliability" being edited by Brian Randell and scheduled for publication by Cambridge University Press, 1978.

3. Martin L. Shooman received the 1977 Annual Reliability Award of the IEEE Reliability Society on January 18, 1978.

4. Martin L. Shooman "Software Engineering: Reliability, Design, Management," accepted for publication by McGraw-Hill, November 1978.